Developing Efficient Code for Decoding AIS Sentences into Useful Datasets

International Association of Marine Aids to Navigation and Lighthouse Authorities

Kiki Beumer

Supervisor: Omar Eriksson

Contents

1 Introduction **3**

2 Decoding AIS Data **3**

3 AIS message standards **4**

4 NMEA 0183 Message Formats **5**

5 Code Overview **5**

6 Detailed Explanation **5**

6.1 Main 6
6.2 decode_ais_nmea Function 6
6.3 verify_checksum Function 6
6.4 decode_payload Function 7
6.5 decode_ais_message Function 8
6.6 combine_to_datetime Function 13
6.7 get_maneuver_ind Function 13
6.8 get_position_fix_type Function 14
6.9 minute_to_dms_long Function 14
6.10 minute_to_dms_lat Function 15
6.11 binary_to_float Function 15
6.12 binary_to_float Function 15
6.13 get_ship_type Function 16
6.14 get_navigation_status Function 16
6.15 get_region Function 17
6.16 get_destination Function 17
6.17 get_vessel_and_call Function 18

7 Results **19**

8 Limitations **22**

9 Possible future approaches **22**

10 Conclusion **22**

11 References **22**

1 Introduction

The purpose of this code is to decode AIS data transmitted using the NMEA 0183 standard, developed and maintained by the National Marine Electronics Association (NMEA) to establish interface standards for marine electronic equipment.

To extract useful information from these AIS sentences, we decode them into an understandable dataframe. Existing AIS decoding codes often require significant time; for example, decoding 1 million sentences can take approximately 3-4 hours. In contrast, this code achieves the same task in significantly less time, approximately 5-9 minutes for decoding approximately 6.5 million sentences.

This decoding process is simplified in that it focuses solely on extraction without filtering or encoding data, which may be considered a limitation. The code serves as a foundational framework intended for future expansion and enhancement. Additional functionalities can be incorporated as necessary. The code to decode the AIS messages is included in the reference section and can be found on Github.

This report provides an overview of the code's functions, to give an understanding of its framework. Its goal is to encourage and assist users in extending and customizing the code to meet specific needs.

2 Decoding AIS Data

AIS messages are commonly transmitted using the NMEA 0183 standard, developed and maintained by the National Marine Electronics Association to establish interface standards for marine electronic equipment. NMEA 0183 is a standard framework for exchanging marine instrument data between various onboard equipment.
An example of an AIS sentence is as follows:

```
!AIVDM,2,1,3,B,55P5TL01VIaAL@7WKO@mBplU@<PDhh000000001S;AJ::4A80?4i@E53,0*3E
!AIVDM,2,2,3,B,1@0000000000000,2*55
```

Figure 1: Example of a multi fragment sentence

Field 1, Format: !AIVDM, identifies this as an AIVDM packet. Other packets are i.a. AIVDO, BSVDM, BSVDO.

Field 2, message count: Total number of messages, sometimes AIS messages are split over several messages due to size limitation.

Field 3, message number: The fragment number of this sentence. It will be one-based. A sentence with a fragment count of 1 and a fragment number of 1 is complete in itself.

Field 4, sequence ID: The message ID if message count is larger than 1.

Field 5, radio channel code: AIS uses the high side of the duplex from two VHF radio channels: AIS Channel A is 161.975Mhz (87B); AIS Channel B is 162.025Mhz (88B).

Field 6, payload: This is the AIS data itself encoded in six bit ASCII. Information such as MMSI number, navigation status, longitude, latitude and speed.

Field 7, size: The number of bits required to fill the data.

Field 8, checksum: The checksum is needed to verify sentence integrity.

3 AIS message standards

NMEA 0183

NMEA 0183 sentences typically start with a dollar sign ($) and an exclamation mark (!) for AIS messages. They are ASCII text sentences that follow a specific structure.
Example: !AIVDM,1,1,,A,13aG?P0000P@¿VPRdLwsv0nN0D1K,0*62
Other NMEA Sentences start with $, followed by a talker ID and a message type.
Example: $GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47

IEC 61162

IEC 61162 follows the same conventions for AIS data as NMEA 0183 and often uses the same sentence structure with $ or !

ITU-R M.1371

ITU-R M.1371 does not define the format of the messages directly, but the AIS data encapsulated within this standard often gets transmitted via NMEA 0183 or similar formats when interfacing with devices.

VDES

VDES messages are more complex and can include both AIS and additional data types. The identification of VDES messages typically happens through context in VHF communications and through their extended message types.

NMEA 2000

NMEA 2000 uses a completely different approach, relying on Controller Area Network technology. It does not use simple ASCII sentences, but rather binary messages called Parameter Group Numbers.
Example: 129038 for AIS Class A position reports.

This code focuses on decoding the AIS messages with NMEA 0183. If you need to decode messages using a different standard, you can incorporate it by adding a function.

4 NMEA 0183 Message Formats

AIVDM:
AIVDM messages are typically received from AIS transponders on vessels. These are the standard messages used by Class A AIS transceivers, which are required for large vessels. AI stands for AIS, and VDM stands for VHF Data-link Message. It is the most common type used for transmitting AIS data in NMEA 0183 format.

BIVDM:
BIVDM messages are typically received from AIS base stations or other non-shipborne sources. These messages are used by Class B AIS transceivers, which are typically found on smaller vessels and pleasure crafts. BI indicates a different source (Base Station or Class B device), and VDM stands for VHF Data-link Message. Used for transmitting AIS data in NMEA 0183 format from base stations or Class B transceivers.

Proprietary AIS message:
A proprietary NMEA 0183 AIS message typically starts with a special prefix, such as $P followed by a manufacturer code and then the data sentence. What the remaining fields of this AIS message mean depends on the context provided by the manufacturer. This typically includes the specifications or the documentation for their proprietary sentences.
The proprietary AIS messages utilize the "Datafield" columns (shown in figure 5).

5 Code Overview

This code contains several functions, each responsible for decoding a different part of the AIS sentence payload. These functions either format the data correctly or assign a value to the corresponding information.

The process begins by verifying the checksum of the AIS sentence to ensure data integrity. Next, all relevant fields are extracted from the sentence, such as the packet type (field 1) and radio channel (field 5). After this, the payload is extracted, decoded, and a binary string is returned. Based on the message type, specific sections of this binary string are extracted.

From this binary string, information such as the rate of turn, longitude, speed over ground, and other pertinent data is retrieved. Additional functions are then called to interpret this decoded data, providing it with meaningful context.

6 Detailed Explanation

This chapter will provide a detailed exploration of the purpose and functionality of each function.

6.1 Main

The main script imports pandas to display the data as a dataframe. The data is imported from a text file, with sentences extracted line by line. The decode_ais_nmea function is called for each sentence, and the decoded messages are stored in a list. Finally, this list is displayed as a dataframe.

```python
import pandas as pd

# Initialize a list
decoded_messages = []

# Open the file in read mode
with open("Dataset001.ais.txt", 'r') as file:
    for line in file:
        nmea_sentence = line.strip()

        decoded_message = decode_ais_nmea(nmea_sentence)
        decoded_messages.append(decoded_message)

# Create a DataFrame from the list of decoded messages
df = pd.DataFrame(decoded_messages)
df
```

Listing 1: Python code for main

6.2 decode_ais_nmea Function

This function verifies the integrity of the checksum. The different fields described previously are stored in a list. From this list the payload is extracted and converted to a binary string. decode_ais_message finally decodes the binary string into meaningful information.

```python
def decode_ais_nmea(nmea_sentence):
    if not verify_checksum(nmea_sentence):
        return {'Error': 'Checksum mismatch'}

    fields = nmea_sentence.split(',')
    payload = fields[5]
    binary_payload = decode_payload(payload)
    decoded_message = decode_ais_message(binary_payload, fields)

    return decoded_message
```

Listing 2: Python code for decode_ais_nmea function

6.3 verify_checksum Function

As mentioned previously, the checksum verifies the accuracy of the AIS sentence. In this text file, a timestamp appears before the AIS sentence, so the timestamp is extracted first, followed by the checksum and then the '!'. The remaining part of the sentence is iterated over, performing the XOR operation to calculate the checksum. This function returns whether the expected and calculated checksums are equal.

```python
def verify_checksum(nmea_sentence):
    """Verify the NMEA sentence checksum."""
    time, ais_sentence = nmea_sentence.split(' ', 1)
    nmea_data, checksum = ais_sentence.split('*')
    nmea_data = nmea_data.lstrip('!')

    calc_checksum = 0
    for char in nmea_data:
        calc_checksum ^= ord(char) #returns integer unicode(subset of ASCII)

    return int(checksum, 16) == calc_checksum
```

Listing 3: Python code for `verify_checksum` function

6.4 `decode_payload` Function

The sole goal of this function is to convert the six bit ASCII payload into binary. The format(value, ’06b’) statement is used to format an integer value as a binary string.

```python
def decode_payload(payload):
    six_bit_ascii = {
        '0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7,
        '8': 8, '9': 9, ':': 10, ';': 11, '<': 12, '=': 13, '>': 14, '?': 15,
        '@': 16, 'A': 17, 'B': 18, 'C': 19, 'D': 20, 'E': 21, 'F': 22, 'G': 23,
        'H': 24, 'I': 25, 'J': 26, 'K': 27, 'L': 28, 'M': 29, 'N': 30, 'O': 31,
        'P': 32, 'Q': 33, 'R': 34, 'S': 35, 'T': 36, 'U': 37, 'V': 38, 'W': 39,
        '`': 40, 'a': 41, 'b': 42, 'c': 43, 'd': 44, 'e': 45, 'f': 46, 'g': 47,
        'h': 48, 'i': 49, 'j': 50, 'k': 51, 'l': 52, 'm': 53, 'n': 54, 'o': 55,
        'p': 56, 'q': 57, 'r': 58, 's': 59, 't': 60, 'u': 61, 'v': 62, 'w': 63
    }

    binary_str = ''
    for char in payload:
        if char in six_bit_ascii:
            value = six_bit_ascii[char]
            binary_str += format(value, '06b')

    #Add else statement here if needed

    return binary_str
```

Listing 4: Python code for `decode_payload` function

6.5 `decode_ais_message` Function

The parameters of this function are the previously obtained binary string and the `fields` list. Based on the message type, the binary string is decoded. Not all variables are used for each message type, so the variables are first initialized to ensure they appear neatly in the dataset. Additional information is assigned to these variables as required. Before decoding, `if len(binary_str) < n:` checks whether the binary string meets the length requirements to avoid incorrect values. All decoded information is then returned. The proprietary packet type provides specific data based on the context defined by the manufacturer. Consequently, there are five proprietary message columns used to display the data contained within the message.

```python
from datetime import datetime

def decode_ais_message1(binary_str, fields):

    #Columns for all message types
    time_format, packet_type = fields[0].split(' ', 1)
    #time
    time = datetime.strptime(time_format, "%Y-%m-%dT%H:%M:%S.%fZ")

    #Proprietary packet type
    if packet_type.startswith("$"):
        return {
            'Timestamp': time,
            'Packet Type': packet_type.lstrip('$'),
            'Prop message 1': fields[1],
            'Prop message 2': fields[2],
            'Prop message 3': fields[3],
            'Prop message 4': fields[4],
            'Prop message 5': fields[5]
        }

    message_type = int(binary_str[0:6], 2)
    repeat_ind = int(binary_str[6:8], 2)
    mmsi = check_mmsi(int(binary_str[8:38], 2))
    channel = fields[4]

    #Columns different per message type
    ais_version = 0
    imo = 0
    call_sign = 0
    vessel_name = 'NaN'
    ship_type = get_ship_type(0)
    a = 0
    b = 0
    c = 0
    d = 0
    eta = 'NaN'
    draught = 0
    destination = 'NaN'
    nav_status = 'NaN'
    rot = float('nan')
    sog = float('nan')
    cog = 'NaN'
```

```
    position_acc = 'NaN'
    long = 0
    lat = 0
    heading = float('nan')
    radio_status = 0
    sotdma = 0
    pos_fix_epfd = get_position_fix_type(0)
    maneuver = get_maneuver_ind(0)

################type 1,2,3 ################
    if message_type in [1, 2, 3]:
        #rate of turn
        if len(binary_str) < 50:
            rot = float('nan')
        else:
            rot= int(binary_str[42:50], 2)
            if (1 <= rot <= 126):
                rot = int(rot / 4.733) ** 2
            elif (-126 <= rot <= -1):
                rot = (int(rot / 4.733) ** 2)*-1
            #elif rot == -127 or rot== 127:
                #rot = 'Turn more than 5deg/sec'
            #elif rot == 0:
               # rot = 'Not turning'


        #speed over ground
        if len(binary_str) < 60:
            sog = float('nan')
        else:
            sog = int(binary_str[50:60], 2)*0.1
            if sog == 102.3:
                sog = float('nan')
            elif sog == 102.2:
                sog = '102.2 knots or higher'

        #position accuracy
        if len(binary_str) < 61:
            position_acc = float('nan')
        else:
            position_acc = int(binary_str[60:61], 2)
            if position_acc == 1:
                position_acc = '<10m'
            elif position_acc ==0:
                position_acc = '>10m'


        #longitude
        if len(binary_str) < 89:
            long = float('nan')
        else:
            long = minute_to_dms_long(binary_str[61:89])

        #latitude
        if len(binary_str) < 116:
            lat = float('nan')
        else:
```

```
            lat = minute_to_dms_lat(binary_str[89:116])

        #Course over ground
        if len(binary_str) < 128:
            cog = float('nan')
        else:
            cog = int(binary_str[116:128], 2)
            if cog == 3600:
                cog = 'NaN'

        #True Heading
        if len(binary_str) < 137:
            heading = float('nan')
        else:
            heading = int(binary_str[128:137], 2)
            if heading == 511:
                heading = float('nan')

        #Radio status
        if len(binary_str) < 168:
            radio_status = float('nan')
        else:
            radio_status = int(binary_str[149:168], 2)

        #Navigation status
        if len(binary_str) < 42:
            nav_status = float('nan')
        else:
            nav_status = get_navigation_status(int(binary_str[38:42], 2))

        #Maneuver
        if len(binary_str) < 42:
            maneuver = get_maneuver_ind(0)
        else:
            maneuver = get_maneuver_ind(int(binary_str[38:42], 2))

    ###############type 4 ###############

    elif message_type == 4:

        #longitude
        if len(binary_str) < 107:
            long = float('nan')
        else:
            long = minute_to_dms_long(binary_str[79:107])

        #latitude
        if len(binary_str) < 134:
            lat = float('nan')
        else:
            lat = minute_to_dms_lat(binary_str[107:134])

        #Type of EPFD
        if len(binary_str) < 138:
            pos_fix_epfd = get_position_fix_type(0)
        else:
```

```
            pos_fix_epfd = get_position_fix_type(int(binary_str
    [134:138], 2))

        #SOTDMA state (radio)
        if len(binary_str) < 168:
            radio_status = float('nan')
        else:
            radio_status = int(binary_str[149:168], 2)

        #ETA
        if len(binary_str) < 78:
            eta = float('nan')
        else:
            eta = combine_to_datetime(binary_str)


    ###### type 5#####
    elif message_type == 5:

        #AIS version
        if len(binary_str) < 40:
            ais_version = float('nan')
        else:
            ais_version= int(binary_str[38:40],2)

        #IMO number
        if len(binary_str) < 70:
            imo = float('nan')
        else:
            imo= int(binary_str[40:70],2)

        #Call Sign
        if len(binary_str) < 112:
            call_sign = float('nan')
        else:
            call_sign= get_vessel_and_call(binary_str[70:112])

        #Vessel name
        if len(binary_str) < 232:
            vessel_name = 'NaN'
        else:
            vessel_name= get_vessel_and_call(binary_str[112:232])

        #Ship type
        if len(binary_str) < 240:
            ship_type = get_ship_type(100)
        else:
            ship_type= get_ship_type(int(binary_str[232:240],2))

        #Dimension to Bow
        if len(binary_str) < 249:
            a = float('nan')
        else:
            a = int(binary_str[240:249], 2)

        #Dimension to Stern
        if len(binary_str) < 258:
```

```
            b = float('nan')
        else:
            b = int(binary_str[249:258], 2)

        #Dimension to Port
        if len(binary_str) < 264:
            d = float('nan')
        else:
            d = int(binary_str[258:264], 2)

        #Dimension to Starboard
        if len(binary_str) < 270:
            c = float('nan')
        else:
            c = int(binary_str[264:270], 2)

        #Position fix type
        if len(binary_str) < 274:
            pos_fix_epfd = get_position_fix_type(0)
        else:
            pos_fix_epfd = get_position_fix_type(int(binary_str[270:274], 2))

        #Draught
        if len(binary_str) < 302:
            draught = float('nan')
        else:
            draught = int(binary_str[294:302], 2)/10

        #Destination
        if len(binary_str) < 422:
            destination = 'NaN'
        else:
            destination = get_destination(binary_str[302:422], 2)

    #Region
    if lat == 0 or long == 0:
        region = 'NaN'
    else:
        region = get_region(long, lat)

    return {
            'Timestamp': time,
            'Packet Type': packet_type.lstrip('!'),
            'Channel': channel,
            'Message Type': message_type,
            'MMSI': mmsi,
            'Navigation Status': nav_status,
            'Repeat Indicator':repeat_ind,
            'IMO': imo,
            'ROT': rot,
            'SOG': sog,
            'COG': cog,
            'Position Accuracy': position_acc,
            'Longitude':long,
            'Latitude':lat,
            'Region':region,
```

```python
            'Vessel name': vessel_name,
            'Ship type': ship_type,
            'True Heading': heading,
            'Radio status': radio_status,
            'Destination': destination,
            'Maneuver Indicator': maneuver,
            'Draught': draught,
            'Position fix type': pos_fix_epfd,
            'Call sign': call_sign,
            'ETA': eta,
            'A':a,
            'B': b,
            'C': c,
            'D': d
        }
```

Listing 5: Python code for decode_ais_message function

6.6 combine_to_datetime Function

The combine_to_datetime function converts a segment of a binary string into a datetime object. It extracts specific date and time components (year, month, day, hour, minute, and second) from the binary string by decoding predefined bit positions into integers. These extracted components are then combined using the datetime constructor from the datetime library.

```python
from datetime import datetime
def combine_to_datetime(binary_str):
    return datetime(year=int(binary_str[38:52], 2),
                    month=int(binary_str[52:56], 2),
                    day=int(binary_str[56:61], 2),
                    hour=int(binary_str[61:66], 2),
                    minute=int(binary_str[66:72], 2),
                    second=int(binary_str[72:78], 2))
```

Listing 6: Python code for combine_to_datetime function

6.7 get_maneuver_ind Function

This function translates a maneuver indicator from AIS data into a human-readable format. It uses a dictionary, maneuver_decode, to map integer values to descriptive strings based on the AIS standard. The function takes an integer maneuver as input and returns the corresponding description, such as "Not available (default)" or "No special maneuver". If the input value is not in the dictionary, it returns "Unknown status".

```python
def get_maneuver_ind(maneuver):
    # Define the navigation statuses based on AIS standard
    maneuver_decode = {
        0: "Not available (default)",
        1: "No special maneuver",
        2: "Special maneuver(such as regional passing arrangement)"
    }
```

```

    # Return the corresponding navigation status
    return maneuver_decode.get(maneuver, "Unknown status")
```

Listing 7: Python code for `get_maneuver_ind` function

6.8 `get_position_fix_type` Function

This function translates a position fix type from AIS data into a readable format. It uses again a dictionary, position_fix_decode, to map integer values to descriptive strings based on the AIS standard. The function takes an integer `pos_fix_epfd` as input and returns the corresponding description, such as "Undefined (default)", "Chayka" or "GPS". If the input value is not in the dictionary, it returns "Unknown status".

```
def get_position_fix_type(pos_fix_epfd):
    # Define the navigation statuses based on AIS standard
    position_fix_decode = {
        0: "Undefined (default)",
        1: "GPS",
        2: "GLONASS",
        3: "Combined GPS/ GLONASS",
        4: "Loran-C",
        5: "Chayka",
        6: "Integrated navigation system",
        7: "Surveyed",
        8: "Galileo",
        9: "Reserved",
        10: "Reserved",
        11: "Reserved",
        12: "Reserved",
        13: "Reserved",
        14: "Reserved",
        15: "Internal GNSS"
    }
    # Return the corresponding navigation status
    return position_fix_decode.get(pos\_fix_epfd, "Unknown status")
```

Listing 8: Python code for `get_position_fix_type` function

6.9 `minute_to_dms_long` Function

This function decodes the binary string segment into DMS (degrees, minutes, seconds). It first converts the binary string into a float with the `binary_to_float`. That value is divided by 600000.0 to obtain the total minutes, as specified by the AIS standard. We can extract the degrees as the integer part of the total minutes. The remaining fractional minutes are converted to minutes and seconds. However, the function does not perform these calculation as it is more efficient to work with a float rather than an object for analysis. The function returns the longitude of the vessel.

```
def minute_to_dms_long(value):
```

```
    # Convert from thousandths of a minute to minutes
    float_value = binary_to_float(value)
    total_minutes = float_value / 600000.0

    return total_minutes
```

Listing 9: Python code for `minute_to_dms_long` function

6.10 minute_to_dms_lat Function

This function converts the binary string into the latitude similarly as the function mentioned above. However, the latitude ensures the correct negative values if `total_minutes` is larger than 90 degrees.This function returns the latitude.

```
def minute_to_dms_lat(value):
    # Convert from thousandths of a minute to minutes
    float_value = binary_to_float(value)
    total_minutes = float_value / 600000.0

    if total_minutes > 90.0:
        total_minutes = 90- total_minutes

    return total_minutes
```

Listing 10: Python code for `minute_to_dms_lat` function

6.11 binary_to_float Function

To correctly convert the binary string into a float to compute the longitude and latitude, this function is needed.

```
from ast import literal_eval

def binary_to_float(float_str):

    if (float_str)[0] == '-':
        float_str = f"-0b{float_str[1:]}"
    else :
        float_str = f"0b{float_str[1:]}"

    result = float(literal_eval(float_str))

    return result
```

Listing 11: Python code for `binary_to_float` function

6.12 binary_to_float Function

This function checks if the MMSI number is valide. If the MMSI number is null, contains of only zeros, does not have length 7/ 9 or is a consecutive number, then it is not valide. This function checks for any of these possibilities.

```python
from ast import literal_eval

def check_mmsi(mmsi):
    # checks if not null
    try:
        mmsi = str(int(mmsi))
    except:
        return 0, False

    # check if the length is 7 or 9
    if len(mmsi) not in {7, 9}:
        return 0, False

    # should not contain all same digits 000000000
    if len(set(mmsi)) == 1:
        return 0, False

    # should not be consecutive eg: 123456789
    if [int(i) for i in mmsi] == list(range(int(min(mmsi)), int(max(
    mmsi)) + 1)):
        return 0, False

    return int(mmsi)
```

Listing 12: Python code for `binary_to_float` function

6.13 `get_ship_type` Function

This function operates on the same principle as previously described. For a detailed explanation of its logic, please refer to the earlier section.

```python
def get_ship_type(ship_type):
    ship_type_decode = {
        0: "Not available (default)",
        1: "Reserved for future use",
        2: "Reserved for future use",
        3: "Reserved for future use",
        ...
        98: "Other Type, Reserved for future use",
        99: "Other Type, no additional information"
    }

    # Return the corresponding ship type
    return ship_type_decode.get(ship_type, "Unknown ship type")
```

Listing 13: Python code for `get_ship_type` function

6.14 `get_navigation_status` Function

This function operates on the same principle as previously described. For a detailed explanation of its logic, please refer to the earlier section.

```python
def get_navigation_status(nav_status):
    # Define the navigation statuses based on AIS standard
    nav_status_decode = {
```

```python
        0: "Underway using engine",
        1: "At anchor",
        2: "Not under command",
        3: "Restricted manoeuverability",
        4: "Constrained by her draught",
        5: "Moored",
        6: "Aground",
        7: "Engaged in fishing",
        8: "Underway sailing",
        9: "Reserved for future amendment of Navigational Status for HSC",
        10: "Reserved for future amendment of Navigational Status for WIG",
        11: "Power-driven vessel towing astern (regional use)",
        12: "Power-driven vessel pushing ahead or towing alongside (regional use)",
        13: "Reserved for future use",
        14: "AIS-SART is active",
        15: "Undefined (default)"
    }

    # Return the corresponding navigation status
    return nav_status_decode.get(nav_status, "Unknown status")
```

Listing 14: Python code for `get_navigation_status` function

6.15 `get_region` Function

After ensuring the validity of the data, an additional column is added; `Region`. This column is calculated with the function below and the Maidenhead library. This takes the longitude, latitude, and the level of precision and returns the region using the IARU Grid Locator system. Figure 2 displays the grid.

```python
import maidenhead as mh

def get_region(long, lat):
    try:
        region = mh.to_maiden(lat, long, 2)
    except:
        region = "NaN"

    return region
```

Listing 15: Python code for `get_region` function

6.16 `get_destination` Function

For the destination, UN/LOCODE and ERI terminal codes should be used.Therefore, we can use this function to convert the binary string into UN/LOCODE and ERI terminal codes. The binary string needs to be separated into 20 six bit chunks.

```python
def get_destination(binary_str):
    # AIS 6-bit character set mapping
```

```
    ais_charset = '@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_ !"#$%&\'()
    *+,-./0123456789:;<=>?'

    # Convert the binary string to text using 6-bit chunks
    text = ''
    for i in range(0, len(binary_str), 6):
        six_bit_group = binary_str[i:i+6]
        decimal_value = int(six_bit_group, 2)
        text += ais_charset[decimal_value]

    # Extract the codes assuming a fixed format: first 10 characters
    for UN/LOCODEs and next 5 for ERI code
    un_locode_1 = text[:5]
    un_locode_2 = text[5:10]
    eri_code = text[10:15]

    # Combine all three codes into a single string
    combined_codes = un_locode_1 + un_locode_2 + eri_code

    return combined_codes
```

Listing 16: Python code for get_destination function

6.17 `get_vessel_and_call` Function

Both the call sign and the vessel name can be decoded using a function similar to `get_destination`. However, this function doesn't need to split the binary string into three parts. Some vessel names may have trailing '@' characters, which can be removed using `.rstrip('@')` to ensure correct display.

```
def get_vessel_and_call(binary_str):
    # AIS 6-bit character set mapping
    ais_charset = '@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_ !"#$%&\'()
    *+,-./0123456789:;<=>?'

    # Convert the binary string to text using 6-bit chunks
    text = ''
    for i in range(0, len(binary_str), 6):
        six_bit_group = binary_str[i:i+6]
        decimal_value = int(six_bit_group, 2)
        text += ais_charset[decimal_value]

    # Extract the codes from the fixed format
    un_locode_1 = text[:5]
    un_locode_2 = text[5:10]
    eri_code = text[10:15]

    # Combine all three codes into a single string
    combined_codes = un_locode_1 + un_locode_2 + eri_code

    return combined_codes
```

Listing 17: Python code for get_vessel_and_call function

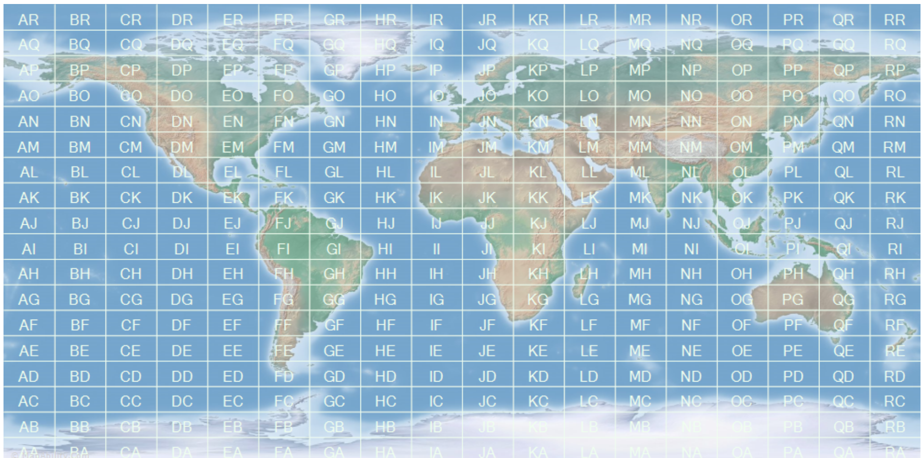

Figure 2: Grid Locator

7 Results

The following figures show the resulting dataset.

Timestamp	Packet Typ	Channel	Message T	MMSI	Navigation	Repeat Ind	IMO	ROT	SOG	COG	Position Ac	Longitude	Latitude
17:51.0	BSVDM	A	1	3.53E+08	Underway	0	0	0	12.5	1728	>10m	152.1202	-10.1655
17:51.3	BSVDO	A	4	2579991	NaN	0	0			NaN	NaN	152.1354	-10.5933
17:52.5	BSADS												
17:52.7	BSVDM	A	1	4.32E+08	Underway	0	0	0	11.3	3501	>10m	152.1153	-10.5118
17:53.5	PSTXI												
17:55.0	BSVDM	B	1	5.38E+08	Underway	0	0	1	16.4	1400	>10m	151.9924	-11.2176
17:55.9	BSVDM	B	1	4.77E+08	Underway	0	0	0	14.5	1320	>10m	151.8351	-11.4112
18:00.5	BSVDM	B	1	3.53E+08	Underway	0	0	0	12.5	1726	>10m	152.1203	-10.1649
18:01.2	BSVDO	B	4	2579991	NaN	0	0			NaN	NaN	152.1354	-10.5933
18:01.8	BSVDM	A	3	5.38E+08	Underway	0	0	1	16.4	1401	>10m	151.9928	-11.2171
18:02.5	BSVDM	B	1	4.32E+08	Underway	0	0	0	11.3	3504	>10m	152.1152	-10.5123
18:03.5	PSTXI												
18:08.2	BSVDM	B	1	4.77E+08	Underway	0	0	0	14.5	1322	>10m	151.8357	-11.4107
18:11.2	BSVDO	A	4	2579991	NaN	0	0			NaN	NaN	152.1354	-10.5933
18:11.3	BSVDM	A	1	3.53E+08	Underway	0	0	0	12.5	1725	>10m	152.1204	-10.1644
18:12.8	BSVDM	A	1	4.32E+08	Underway	0	0	0	11.3	3502	>10m	152.1151	-10.5128
18:13.5	PSTXI												
18:13.8	BSVDM	A	1	5.38E+08	Underway	0	0	0	16.4	1400	>10m	151.9933	-11.2164
18:14.0	BSVDM	A	1	4.77E+08	Underway	0	0	0	14.5	1323	>10m	151.836	-11.4105
18:21.2	BSVDO	B	4	2579991	NaN	0	0			NaN	NaN	152.1354	-10.5933
18:21.3	BSVDM	B	1	4.32E+08	Underway	0	0	0	11.3	3497	>10m	152.115	-10.5133
18:22.0	BSVDM	B	1	3.53E+08	Underway	0	0	0	12.4	1721	>10m	152.1205	-10.1637
18:23.5	PSTXI												
18:25.2	BSVDM	A	1	4.77E+08	Underway	0	0	253	14.5	1325	>10m	151.8366	-11.4099
18:31.2	BSVDO	A	4	2579991	NaN	0	0			NaN	NaN	152.1354	-10.5933

Figure 3: Output csv file

Region	Vessel nan	Ship type	True Headi	Radio statu	Destinatio	Maneuver	Draught	Position fi	Call sign	ETA	A	B	C
QH69	NaN	Not availat	170	81925	NaN	Not availat	0	Undefined	0		0	0	0
QH69	NaN	Not available (default		114692	NaN	Not availat	0	Internal GI	0	#########	0	0	0
QH69	NaN	Not availat	350	2200	NaN	Not availat	0	Undefined	0		0	0	0
QH58	NaN	Not availat	139	81928	NaN	Not availat	0	Undefined	0		0	0	0
QH58	NaN	Not availat	129	2257	NaN	Not availat	0	Undefined	0		0	0	0
QH69	NaN	Not availat	170	2289	NaN	Not availat	0	Undefined	0		0	0	0
QH69	NaN	Not available (default		114692	NaN	Not availat	0	Internal GI	0	#########	0	0	0
QH58	NaN	Not availat	139	24355	NaN	Not availat	0	Undefined	0		0	0	0
QH69	NaN	Not availat	350	20016	NaN	Not availat	0	Undefined	0		0	0	0
QH58	NaN	Not availat	129	49158	NaN	Not availat	0	Undefined	0		0	0	0
QH69	NaN	Not available (default		114692	NaN	Not availat	0	Internal GI	0	#########	0	0	0
QH69	NaN	Not availat	170	114693	NaN	Not availat	0	Undefined	0		0	0	0
QH69	NaN	Not availat	350	67101	NaN	Not availat	0	Undefined	0		0	0	0
QH58	NaN	Not availat	139	20016	NaN	Not availat	0	Undefined	0		0	0	0
QH58	NaN	Not availat	129	34380	NaN	Not availat	0	Undefined	0		0	0	0
QH69	NaN	Not available (default		114692	NaN	Not availat	0	Internal GI	0	#########	0	0	0
QH69	NaN	Not availat	351	20016	NaN	Not availat	0	Undefined	0		0	0	0
QH69	NaN	Not availat	170	81925	NaN	Not availat	0	Undefined	0		0	0	0
QH58	NaN	Not availat	129	2264	NaN	Not availat	0	Undefined	0		0	0	0
QH69	NaN	Not available (default		98314	NaN	Not availat	0	Internal GI	0	#########	0	0	0

Figure 4: Output csv file

D	Prop mess	Prop mess	Prop mess	Prop mess	Prop message 5
0					
0					
	STX56485	71220	V	0	I
0					
	INFO	2	0	0	1
0					
0					
0					
0					
0					
0					
	INFO	2	0	0	1
0					
0					
0					
0					
	INFO	2	0	0	1
0					
0					
0					
0					
0					
	INFO	2	0	0	1
0					
0					

Figure 5: Output csv file

8 Limitations

Decoding proprietary messages is challenging, because their format and context vary by manufacturer. For now, these messages have been included under the "Datafields" section. Additionally, the destination was decoded according to the requirements, but in this dataset, all decoded values were "NaN."
The dataset contains various message types, each providing different information. As a result, many columns contain "NaN" values because not all message types populate every column. Currently, the code only handles message types 1, 2, 3, 4, and 5, as these are the most commonly transmitted. If needed, other message types can be added later.

9 Possible future approaches

In future approaches, we should ensure that all AIS message types are accounted for, with particular emphasis on Message Type 5. This type is needed for visualizing voyages in IWRAP. It will simplify data analysis, making it more effective than working with large csv files. Making IWRAP more intuitive with promote the use of the application to help analyse the data. By incorporating as many columns from the csv files as possible, we can get an even better insight into the voyages.
Additionally, we should consider incorporating extra calculations or conditioning to add columns that provide more meaningful information, similar to the region column included in the report. A closer examination of proprietary messages, especially timestamps, will offer more insights as well.
Implementing a function to decode various formats and standards will also be beneficial, although this will require considerable time. Overall, the goal is to refine data decoding processes. Making AIS data analysis and utilization more efficient and meaningful, will ultimately improve voyage planning and optimization.

10 Conclusion

AIS tracking data offers opportunities to advance marine transportation and safety. By increasing the speed of decoding AIS messages, we can process larger datasets, enabling more accurate and comprehensive analyses due to the larger sample size, which provides a better overall perspective. Additional code may still be required to meet specific needs.

11 References

IALA. Retrieved from:
`https://www.iala-aism.org/`

AIS message decoder on Github. Retrieved from:
https://github.com/kikibeumer/Data-Quality-analysis-if-AIS-dataset/blob/main/AIS_Message_Decoder_Kiki_Beumer.ipynb
Danish Maritime Authority - AIS. Retrieved from:
https://github.com/dma-ais
Automatic identification system. Retrieved from:
https://en.wikipedia.org/wiki/Automatic_identification_system
AIS data Danish Maritime Authority. Retrieved from:
https://www.dma.dk/safety-at-sea/navigational-information/ais-data
GH AIS Message Format. Retrieved from:
https://www.iala-aism.org/wiki/iwrap/index.php/GH_AIS_Message_Format
AIVDM/AIVDO protocol decoding. Retrieved from:
https://gpsd.gitlab.io/gpsd/AIVDM.html
pyais. Retrieved from:
https://github.com/M0r13n/pyais/tree/master?tab=readme-ov-file
ais-protocol-decoding. Retrieved from:
https://github.com/doron2402/ais-protocol-decoding
AIRU Maidenhead Grid Locator Retrieved from:
https://www.mapability.com/ei8ic/maps/gridworld.php